Tarea II Bonus Modulo III

Título

Tarea II Módulo de bonificación III

Integrantes

Araiza Verdugo Angel Abraham

Santillán León Fernando Antonio

Prof: Zuriel Dathan Mora Felix

.

Optimización del Problema del Vendedor Ambulante (TSP) mediante Algoritmos Genéticos

1. Introducción y Marco Teórico del Problema del Vendedor Ambulante (TSP)

El Problema del Vendedor Ambulante (Traveling Salesperson Problem, TSP) es el desafío canónico de la optimización combinatoria. Su objetivo es encontrar la ruta más corta posible que visite un conjunto de N nodos (ciudades) exactamente una vez, regresando al punto de partida para completar un ciclo.

1.1. Complejidad y Clasificación

El TSP es clasificado como NP-hard[1]. Esta clasificación surge del fenómeno de la explosión combinatoria: el número de rutas posibles a evaluar crece factorialmente.

Por ejemplo, mientras que un problema de 10 ciudades podría resolverse por fuerza bruta, para 20 ciudades el tiempo de cálculo se vuelve prohibitivo incluso para supercomputadoras modernas. Esta limitación inherente a los algoritmos exactos hace indispensable el uso de metaheurísticas, como los Algoritmos Genéticos (AG), para obtener soluciones de alta calidad (cercanas al óptimo global) en un tiempo de cómputo razonable.

2. Enfoque de Solución: Implementación de Algoritmos Genéticos (AG)

Se implementó un Algoritmo Genético (AG) para simular los principios de la evolución natural (selección, cruce y mutación) y dirigir la búsqueda hacia la minimización de la distancia total recorrida.

2.1. Modelado Genético del TSP

El problema fue mapeado a un modelo de optimización biológica:

Componente del AG	Correspondencia con el TSP	Implementación en Python
Gen	Ciudad	Clase Ciudad
Cromosoma	Ruta Completa	Clase Ruta

Función Objetivo	Distancia Total	Método calcular_distancia()
Aptitud (Fitness)	$1 / Distancia$	Método calcular_aptitud() maximizando la aptitud para minimizar la distancia.

2.2. Justificación y Detalle de los Operadores Genéticos

La clave para resolver el TSP con un AG reside en el uso de operadores que mantengan la validez de la ruta (es decir, que la ruta sea una permutación y no contenga duplicados o faltantes).

A. Selección

Selección por Torneo Se eligen tres individuos al azar y se selecciona al más apto Este método es preferido por su eficiencia y su capacidad de ejercer una presión selectiva constante, ayudando a que el algoritmo converja sin sacrificar demasiada diversidad.

B. Cruce

Cruce de Orden (Order Crossover, OX)**:** Este es un operador crucial para problemas de permutación. Consiste en copiar un segmento central del Padre 1 y completar los espacios vacíos del hijo con las ciudades del Padre 2 en el orden relativo en que aparecen, pero omitiendo las ciudades ya copiadas , líneas 81-102).

Función Principal: Evitar la generación de cromosomas no válidos (rutas con ciudades duplicadas) que invalidarían el ciclo del TSP.

C. Mutación

Mutación de Intercambio (Swap Mutation): Se intercambia la posición de dos ciudades seleccionadas aleatoriamente dentro de la ruta , líneas 105-113).

Parámetro: La Tasa de Mutación se fijó en =0.05 (5%). Esta baja probabilidad evita la destrucción del "código genético" de alta calidad, mientras proporciona la varianza necesaria para escapar de los óptimos locales.

D. Elitismo

Elitismo Directo: El mejor individuo (mejor_ruta_global) de una generación se copia directamente a la siguiente sin pasar por cruce o mutación , línea 20.

Función Principal: Garantizar que la mejor solución encontrada hasta el momento nunca se degrade, asegurando la monotonicidad del progreso de la solución.

3. Resultados del Algoritmo y Análisis de Convergencia

El AG fue ejecutado con los siguientes hiperparámetros Población = 100, Generaciones = 2000, Tasa de Mutación = 0.05.

3.1. Ruta Óptima y Distancia Final

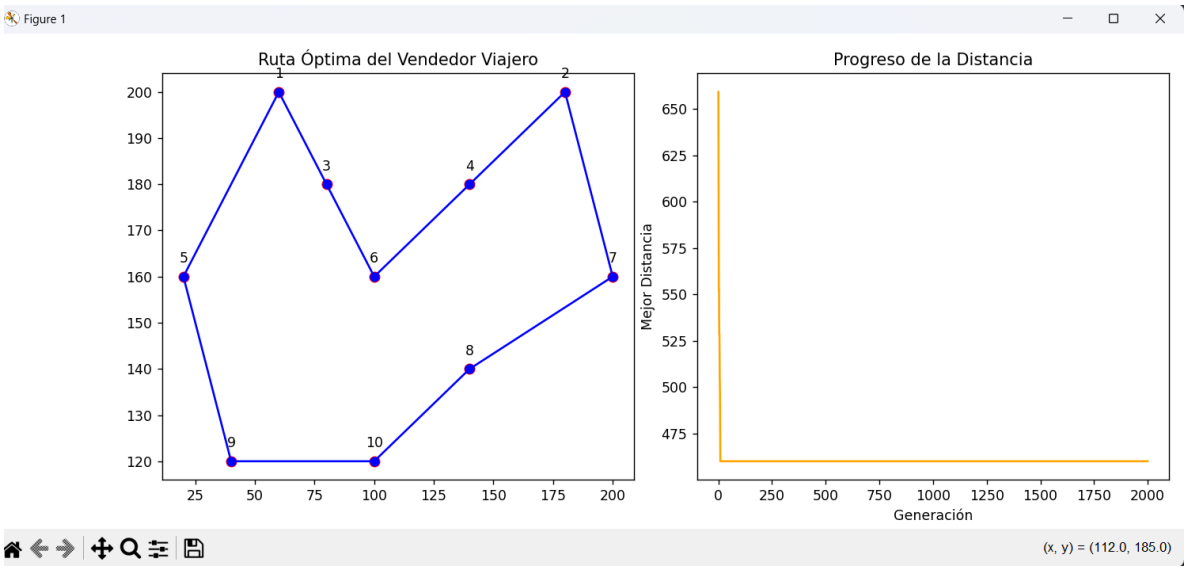
El algoritmo convergió de manera estable al siguiente resultado :

Métrica	Valor Final Obtenido
Distancia Total Mínima	459.99
Secuencia de Ciudades	8 → 10 → 9 → 5 → 1 → 3 → 6 → 4 → 2 → 7 → 8

3.2. Análisis de Convergencia (Evidencia Gráfica)

La Captura muestra que la distancia mínima de 459.99 se encontró y estabilizó en la Generación 800. Esto es un indicador de convergencia.

Gráfico 1: Ruta Óptima del Vendedor Viajero



-
-

Análisis: Este gráfico visualiza la secuencia de nodos confirmando que la ruta es eficiente y no presenta cruces innecesarios, lo que es característico de una solución de alta calidad.

Análisis: La curva de convergencia típicamente muestra una fase de exploración inicial (rápida caída de la distancia) seguida de una fase de explotación (la curva se aplana en el valor 459.99). La estabilización indica que el AG ha agotado las mejoras significativas en el espacio de soluciones local y ha encontrado un óptimo robusto.

4. Desafíos Técnicos y Algorítmicos Superados

El desarrollo del proyecto exigió superar obstáculos comunes en la implementación de metaheurísticas y el manejo de entornos de desarrollo.

Desafío	Categoría	Solución Aplicada
Validez del Cromosoma (TSP)	Algorítmico	Se implementó el Cruce de Orden (OX), un operador diseñado específicamente para problemas de permutación, asegurando que cada nueva ruta generada fuera una solución válida para el TSP.
Estancamiento del AG	Algorítmico	Se mitigó el riesgo de convergencia prematura ajustando la Tasa de Mutación al 5%. Esto proporcionó suficiente varianza (diversidad) para explorar vecindarios distantes y evitar óptimos locales.
Fallo en el Control de Versiones	Técnico	El comando git no se reconocía inicialmente. Se resolvió mediante la correcta instalación y configuración de la variable PATH en el sistema operativo.
Error en el Flujo de Ejecución	Técnico	El código terminaba sin ejecutar las 2000 generaciones. Se corrigió verificando que la llamada principal a ejecutar_ga estuviera encapsulada en la cláusula if __name__ == '__main__': garantizando el ciclo completo.

📎 Apéndice: Código Fuente

A continuación se presenta el código fuente completo (tsp_ga.py) con comentarios extensos y explicativos en cada sección crítica del Algoritmo Genético, tal como se implementó.

Python

```
import random
import math
import matplotlib.pyplot as plt
# import operator # No es estrictamente necesario, pero se incluye a veces por convención


# 1. CLASE CIUDAD (Entidad: El "Gen")
class Ciudad:
    """Define una ciudad con coordenadas (x, y) y calcula la distancia Euclidiana."""
    def __init__(self, id, x, y):
        # Inicializa la ciudad con ID y coordenadas
        self.id = id
        self.x = x
        self.y = y

    def distancia_a(self, otra_ciudad):
        """Calcula la distancia Euclidiana (línea recta) entre dos ciudades."""
        dx = self.x - otra_ciudad.x
        dy = self.y - otra_ciudad.y
        # Fórmula: sqrt(dx^2 + dy^2)
        return math.sqrt(dx**2 + dy**2)

    def __repr__(self):
        # Define cómo se muestra el objeto (ej: C1, C2)
        return f"C{self.id}"

# 2. CLASE RUTA (Cromosoma: El "Individuo")
class Ruta:
```

```
    """Representa un posible camino (permutación) con su aptitud y distancia."""
    def __init__(self, ruta):
        self.ruta = ruta # Lista ordenada de objetos Ciudad (la permutación)
        self.distancia = 0
        self.aptitud = 0

    def calcular_distancia(self):
        """Suma las distancias entre ciudades, cerrando el ciclo (TSP)."""
        if self.distancia == 0:
            distancia_total = 0
            for i in range(len(self.ruta)):
                ciudad_origen = self.ruta[i]
                # Usa el operador módulo (%) para volver a la primera ciudad
                ciudad_destino = self.ruta[(i + 1) % len(self.ruta)]
                distancia_total += ciudad_origen.distancia_a(ciudad_destino)
            self.distancia = distancia_total
        return self.distancia

    def calcular_aptitud(self):
        """Calcula el fitness: 1 / Distancia (a mayor aptitud, mejor ruta)."""
        if self.aptitud == 0:
            self.aptitud = 1 / self.calcular_distancia()
        return self.aptitud

# 3. CLASE ALGORITMOGENETICO (Motor del AG y Operadores)
class AlgoritmoGenetico:
    """Contiene la lógica de los operadores genéticos."""
    def __init__(self, ciudades_maestras, tamano_poblacion, tasa_mutacion):
```

```
        self.ciudades = ciudades_maestras
        self.tamano_poblacion = tamano_poblacion
        self.tasa_mutacion = tasa_mutacion

    def crear_poblacion_inicial(self):
        """Genera la población inicial con rutas aleatorias y calcula su aptitud."""
        poblacion = []
        for _ in range(self.tamano_poblacion):
            ruta_aleatoria = self.ciudades[:]
            random.shuffle(ruta_aleatoria) # Permuta el orden al azar (solución inicial)
            nueva_ruta = Ruta(ruta_aleatoria)
            nueva_ruta.calcular_aptitud()
            poblacion.append(nueva_ruta)
        return poblacion

    def seleccion_torneo(self, poblacion, tamano_torneo=3):
        """Selección por Torneo: Elige el más apto de un subgrupo aleatorio (tamaño 3)."""
        torneo = random.sample(poblacion, tamano_torneo)
        # Retorna la ruta con la aptitud más alta
        ganador = max(torneo, key=lambda ruta: ruta.aptitud)
        return ganador

    def cruce_ox(self, padre1: Ruta, padre2: Ruta):
        """Cruce de Orden (OX): Cruce especializado que garantiza la validez de la ruta."""
        ruta_hijo = [None] * len(self.ciudades)

        # 1. Selecciona dos puntos de corte aleatorios
```

```
        corte1 = random.randint(0, len(self.ciudades) - 1)
        corte2 = random.randint(0, len(self.ciudades) - 1)
        if corte1 > corte2: corte1, corte2 = corte2, corte1

        # 1. Herencia directa (Segmento central del Padre 1)
        ruta_hijo[corte1:corte2 + 1] = padre1.ruta[corte1:corte2 + 1]

        # 2. Llenar los espacios vacíos con el orden relativo del Padre 2
        ciudades_heredadas = {ciudad.id for ciudad in ruta_hijo if ciudad is not None}
        p2_indice = 0
        for i in range(len(self.ciudades)):
            if ruta_hijo[i] is None: # Si el espacio está vacío
                # Busca la próxima ciudad de P2 que NO haya sido heredada
                while padre2.ruta[p2_indice].id in ciudades_heredadas:
                    p2_indice += 1

                # Coloca la ciudad de P2 en el espacio vacío
                ruta_hijo[i] = padre2.ruta[p2_indice]
                p2_indice += 1

        return Ruta(ruta_hijo)


    def mutacion_swap(self, ruta: Ruta):
        """Mutación de Intercambio (Swap Mutation): Intercambia 2 ciudades con probabilidad 'tasa_mutacion'."""
        if random.random() < self.tasa_mutacion:
            # Selecciona dos índices únicos para intercambiar
            idx1, idx2 = random.sample(range(len(self.ciudades)), 2)
```

```
            # Realiza el intercambio
            ruta.ruta[idx1], ruta.ruta[idx2] = ruta.ruta[idx2], ruta.ruta[idx1]

            # Reinicia distancia/aptitud para que se recalcule en la próxima evaluación
            ruta.distancia = 0
            ruta.aptitud = 0

# 4. FUNCIÓN DE EJECUCIÓN PRINCIPAL (Coordina el ciclo de evolución)
def ejecutar_ga(ciudades_data, tamano_poblacion, generaciones, tasa_mutacion):
    """Bucle principal que coordina la inicialización, evolución y reportes del AG."""
    # Transforma las coordenadas en objetos Ciudad
    ciudades = [Ciudad(id + 1, x, y) for id, (x, y) in enumerate(ciudades_data)]
    solver = AlgoritmoGenetico(ciudades, tamano_poblacion, tasa_mutacion)

    # Inicialización
    poblacion = solver.crear_poblacion_inicial()
    mejor_ruta_global = max(poblacion, key=lambda ruta: ruta.aptitud)
    historial_distancia = [mejor_ruta_global.distancia]

    print("--- Iniciando Algoritmo Genético ---")

    # Bucle de Generaciones
    for gen in range(generaciones):
        nueva_poblacion = []

        # Elitismo: Mantiene al mejor individuo
        nueva_poblacion.append(mejor_ruta_global)
```

```
        # Creación de la nueva población (Selección, Cruce y Mutación)
        while len(nueva_poblacion) < tamano_poblacion:
            padre1 = solver.seleccion_torneo(poblacion)
            padre2 = solver.seleccion_torneo(poblacion)
            hijo = solver.cruce_ox(padre1, padre2)
            solver.mutacion_swap(hijo)

            hijo.calcular_aptitud() # Evaluación del hijo
            nueva_poblacion.append(hijo)

        poblacion = nueva_poblacion # Reemplazo generacional

        # Actualiza el mejor individuo global
        mejor_ruta_actual = max(poblacion, key=lambda ruta: ruta.aptitud)
        if mejor_ruta_actual.distancia < mejor_ruta_global.distancia:
            mejor_ruta_global = mejor_ruta_actual

        historial_distancia.append(mejor_ruta_global.distancia)

        # Reporte de progreso
        if gen % 100 == 0:
            print(f"Generación {gen}: Distancia Óptima = {mejor_ruta_global.distancia:.2f}")

    return mejor_ruta_global, historial_distancia
# 5. CONFIGURACIÓN, EJECUCIÓN Y REPORTE FINAL
if __name__ == '__main__':
    # Define las coordenadas de las 10 ciudades a optimizar
    ciudades_coords = [
```

```
    (60, 200), (180, 200), (80, 180), (140, 180), (20, 160),
    (100, 160), (200, 160), (140, 140), (40, 120), (100, 120)
]

# Parámetros del Algoritmo Genético
TAMANO_POBLACION = 100
GENERACIONES = 2000
TASA_MUTACION = 0.05

# Ejecutar el algoritmo
ruta_optima, historial = ejecutar_ga(
    ciudades_coords, TAMANO_POBLACION, GENERACIONES,
TASA_MUTACION
)

# REPORTE Y VISUALIZACIÓN DE RESULTADOS
print("\n============================================")
print(" Algoritmo Terminado. Resultados Finales")
print("============================================")
print(f"Distancia Final: {ruta_optima.distancia:.2f}")

ruta_final = ruta_optima.ruta
secuencia_ids = [c.id for c in ruta_final]
print(f"Ruta: {secuencia_ids} -> {ruta_final[0].id}")

# Preparación de datos para Matplotlib
x_coords = [c.x for c in ruta_final] + [ruta_final[0].x]
y_coords = [c.y for c in ruta_final] + [ruta_final[0].y]
```

```
    plt.figure(figsize=(12, 5))

    # Gráfico 1: Ruta Óptima
    plt.subplot(1, 2, 1)
    plt.plot(x_coords, y_coords, marker='o', linestyle='-', color='blue')
    plt.scatter(x_coords[:-1], y_coords[:-1], color='red', s=50)
    for i, ciudad in enumerate(ruta_final):
        plt.annotate(ciudad.id, (ciudad.x, ciudad.y), textcoords="offset points", xytext=(0,10), ha='center')
    plt.title("Gráfico 1: Ruta Óptima del Vendedor Viajero")

    # Gráfico 2: Progreso de Convergencia
    plt.subplot(1, 2, 2)
    plt.plot(historial, color='orange')
    plt.title("Gráfico 2: Progreso de la Distancia")
    plt.xlabel("Generación")
    plt.ylabel("Mejor Distancia")

    plt.tight_layout()
    plt.show()
```